Paralení programování pro vícejádrové stroje s použitím OpenMP

B4B36PDV – Paralelní a distribuované výpočty

Minulé cvičení:

“Vlákna a jejich synchronizace v C++ 11...”

Minulé cvičení:

“Vlákna a jejich synchronizace v C++ 11...”

Programování vícevláknových aplikací ručně může být dřina. Proč znovu objevovat kolo, když mužeme použít hotové řešení?

Minulé cvičení:

“Vlákna a jejich synchronizace v C++ 11...”

Programování vícevláknových aplikací ručně může být dřina. Proč znovu objevovat kolo, když mužeme použít hotové řešení?

Dnešní menu: OpenMP

Osnova

- Opakování z minulého cvičení
- Úvod do OpenMP
- Paralelní bloky se sdílenou pamětí a synchronizace
- Redukce s OpenMP
- Rozvrhování výpočtu v OpenMP

- Zadání druhé domácí úlohy

Opakování z minulého cvičení

http://goo.gl/a6BEMb

Co je OpenMP?

OpenMP: Přehled

- API pro psání vícevláknových aplikací se sdílenou pamětí
- Sada directiv, proměnných prostředí a rutin pro kompilátor a programátory
- Ulehčuje psaní vícevláknových aplikací v C/C++ a Fortran na většině platform s podporou většiny instrukčních sad a operačních systémů

OpenMP: Přehled

- API pro psání vícevláknových aplikací se sdílenou pamětí
- Sada directiv, proměnných prostředí a rutin pro kompilátor a programátory
- Ulehčuje psaní vícevláknových aplikací v C/C++ a Fortran na většině platform s podporou většiny instrukčních sad a operačních systémů

Jako základní referenční příručku můžete použít

`https://msdn.microsoft.com/en-us/library/tt15eb9t.aspx`

Otestujte si své prostředí

`omp_get_num_procs()`
: Počet procesorů, které OpenMP využívá v době volání funkce

`omp_get_num_threads()`
: Počet vláken, které OpenMP využívá v době volání funkce

`omp_get_max_threads()`
: Maximální počet vláken, které OpenMP může využít

`omp_in_parallel()`
: Vrací nenulovou hodnotu, pokud jsme uvnitř paralelního bloku

`omp_get_nested()`
: Vrací nenulu, pokud je povoleno vnořování paralelních bloků

Otestujte si své prostředí

`omp_get_num_procs()`
: Počet procesorů, které OpenMP využívá v době volání funkce

`omp_get_num_threads()`
: Počet vláken, které OpenMP využívá v době volání funkce

`omp_get_max_threads()`
: Maximální počet vláken, které OpenMP může využít

`omp_in_parallel()`
: Vrací nenulovou hodnotu, pokud jsme uvnitř paralelního bloku

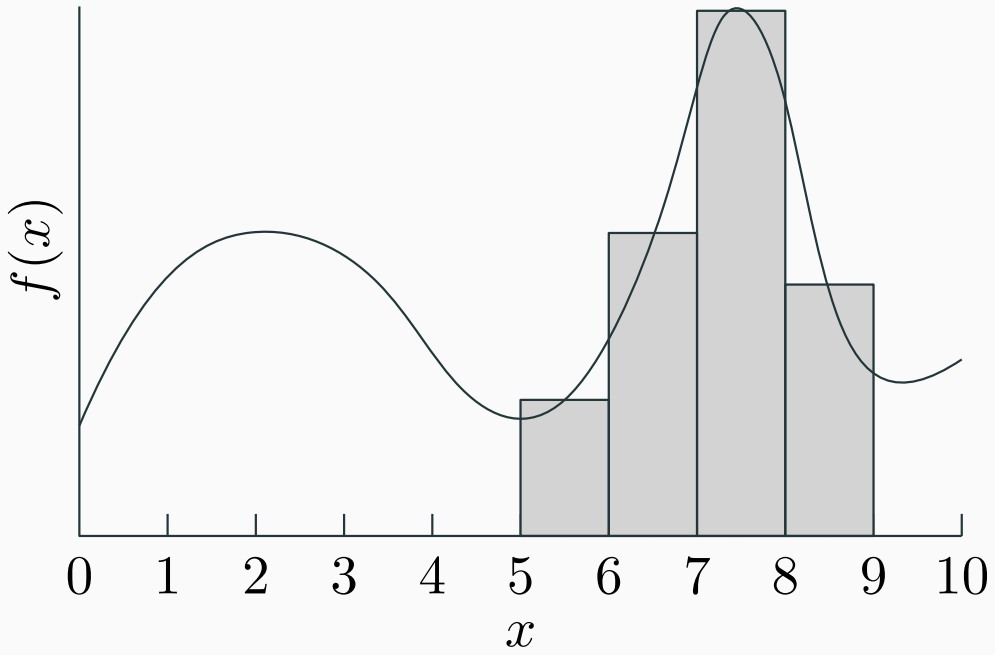
`omp_get_nested()`
: Vrací nenulu, pokud je povoleno vnořování paralelních bloků

Detailní přehled metod s ukázkami na
`https://msdn.microsoft.com/en-us/library/k1h4zbed.aspx`.

Cvičení: Numerická integrace

Numerická integrace

$$\int_5^9 f(x)\,\mathrm{d}x \sim \text{plocha}$$




```
double integrate(
  std::function<double(double)> integrand,
  double a, double step_size, int step_count);
```

Doimplementujte sekvenční verzi numerické integrace

Doimplementujte tělo metody `integrate_sequential` v souboru `integrate.cpp`. Použijte obdélníkovou metodu, kdy jako "výšku" obdélníku použijete hodnotu funkce uprostřed intervalu.

`integrand`
: Funkce, kterou máte za úkol numericky zintegrovat

`a`
: Dolní mez integrálu

`step_size`
: Velikost kroku (šířka obdélníku)

`num_steps`
: Počet kroků (horní mez je `a + step_size * step_count`)

Doimplementujte sekvenční verzi numerické integrace

Doimplementujte tělo metody `integrate_sequential` v souboru `integrate.cpp`. Použijte obdélníkovou metodu, kdy jako "výšku" obdélníku použijete hodnotu funkce uprostřed intervalu.

`integrand`
: Funkce, kterou máte za úkol numericky zintegrovat

`a`
: Dolní mez integrálu

`step_size`
: Velikost kroku (šířka obdélníku)

`num_steps`
: Počet kroků (horní mez je `a + step_size * step_count`)

Jaké problémy budeme mít, pokud budeme chtít tento sekvenční kód paralelizovat?

Alternativy k mutexům a atomickým proměným v OpenMP

#pragma omp parallel

```
int num_threads = 0;
#pragma omp parallel
{
  // Zde jsme vytvorili tym vlaken, ktera vykonavaji
  // nasledujici kod
  num_threads += 1;
}
```

#pragma omp parallel

```
int num_threads = 0;
#pragma omp parallel
{
  // Zde jsme vytvorili tym vlaken, ktera vykonavaji
  // nasledujici kod
  num_threads += 1;
}
```

Jaký bude výsledek?

#pragma omp critical ("mutex")

```
int num_threads = 0;
#pragma omp parallel
{
  // Zde muze byt vice vlaken soucasne...

  #pragma omp critical
  {
    // ..,ale inkrementaci provadi vzdy maximalne
    // jedno vlakno
    num_threads += 1;
  }

  // Zde opet muze byt vice vlaken soucasne
}
```

Doimplementujte metodu `integrate_omp_critical`

Doimplementujte metodu `integrate_omp_critical` v `integrate.cpp`. Využijte k tomu `#pragma omp parallel` a `#pragma omp critical`.

Tip: Po spuštění vláken v bloku `#pragma omp parallel` si můžete napočítat rozsahy indexů, které jednotlivá vlákna budou zpracovávat (viz `decrypt_threads_4` z minulého cvičení). Pro zjištění indexu aktuálního vlákna použijte metodu `omp_get_thread_num()`. Zjistit celkový počet vláken lze pomocí `omp_get_num_threads()`.

#pragma omp atomic

Na minulém cvičení jsme si ukázali, že mutexy mohou být pomalé.
Opravdu pomalé.

→ Jednoduché operace nad jednou proměnnou lze řešit *hardwarovým* zámkem – provedením atomické operace

```
int num_threads = 0;
#pragma omp parallel
{
  #pragma omp atomic
  num_threads += 1;
}
```

Ne všechny operace lze provést atomicky!

Typicky pouze: `x++`, `x--`, `++x`, `--x`
a `x` *OP*`= expr`, kde

$$OP \in \{ +, -, *, /, \&, \hat{}, |, <<, >> \}$$

👍 Pokud kompilátor nemá k dispozici danou atomickou operaci, použije záložní plán: mutex.

Doimplementujte metodu `integrate_omp_atomic`

Doimplementujte metodu `integrate_omp_atomic` v `integrate.cpp`. Místo kritické sekce využijte `#pragma omp atomic`. Jakého zrychlení touto úpravou dosáhneme?

Redukce v OpenMP

Redukce v OpenMP

To samé lze ale udělat elegantněji a efektivněji:

```
int num_threads = 0;
#pragma omp parallel reduction(+:num_threads)
{
  num_threads += 1;
}
```

OpenMP pak zajistí, že se částečné výsledky *lokálních* proměnných `num_threads` po konci bloku posčítají

Následující "operátory" jsou podporované (OpenMP verze 3+):

- Aritmetické: `+`, `*`, `-`, `max`, `min`
- Logické: `&`, `&&`, `|`, `||`, `^`

Doimplementujte metodu `integrate_omp_reduction`

Doimplementujte tělo metody `integrate_omp_reduction` v souboru `integrate.cpp`. Nahraďte `#pragma omp atomic` redukcí.

#pragma omp parallel for

Kód s redukcí lze napsat ještě jednodušeji.

Rozsahy pro vlákna si nemusíme počítat ručně a můžeme práci nechat na OpenMP:

```
double acc = 0.0;

#pragma omp parallel for reduction(+:acc) //schedule(static)
for(int i = 0 ; i < step_count ; i++) {
  const double cx = a + (2*i + 1.0)*step_size/2;
  acc += integrand(cx)*step_size;
}
return acc;
```

Proč při integraci funkce $f(x) = x$
dosahujeme většího zrychlení?

Proč při integraci funkce $f(x) = x$ dosahujeme většího zrychlení?

Výpočet $f(x) = x$ trvá konstantní dobu a práce je tak mezi vlákna rozdělena rovnoměrně.

To neplatí o funkci $f(x) = \int_0^{0.001x^2} \mathbf{sin}(p)\,\mathrm{d}p$, kterou aproximujeme numerickou integrací s proměnlivým počtem kroků.

Doimplementujte metodu `integrate_omp_for_dynamic`

Doimplementujte tělo metody `integrate_omp_for_dynamic`. Statické rozvrhování `schedule(static)` nahraďte dynamickým `schedule(dynamic)`. Jaký má tato volba dopad na rychlost numerické integrace $f(x) = x$ a $f(x) = \int_0^{0.001x^2} \sin(p)\,\mathrm{d}p$?

`#pragma omp parallel for schedule`

Obecná syntaxe (možno použít i další parametry jako např. `reduction`):

```
#pragma omp parallel for schedule(type[,chunk_size])
```

`#pragma omp parallel for schedule`

Obecná syntaxe (možno použít i další parametry jako např. `reduction`):

```
#pragma omp parallel for schedule(type[,chunk_size])
```

`chunk_size` udává minimální velikost bloku, se kterým se plánuje, např:

```
#pragma omp parallel for schedule(dynamic,16)
```

zajistí, že si vlákno po dokončení práce na aktuálním bloku dat řekne o další blok o 16 prvcích.

`#pragma omp parallel for schedule`

Obecná syntaxe (možno použít i další parametry jako např. `reduction`):

```
#pragma omp parallel for schedule(type[,chunk_size])
```

`chunk_size` udává minimální velikost bloku, se kterým se plánuje, např:

```
#pragma omp parallel for schedule(dynamic,16)
```

zajistí, že si vlákno po dokončení práce na aktuálním bloku dat řekne o další blok o 16 prvcích.

- dynamic - vlákna si *dynamicky* alokují bloky, které mají počítat
- guided - *dynamické* plánování, kde se velikost bloků v průběhu výpočtu zmenšuje
- static - každé vlákno má svůj blok přiřazený napevno (když skončí dříve, musí čekat)
- runtime - rozhodnuto za běhu na základě nastavení prostředí (`export OMP_SCHEDULE="dynamic, 100"`)

Zadání druhé domácí úlohy

Paralelní suma vektoru

V 2. domácí úloze si budete moct vyzkoušet, že úspěšnost různých způsobů paralelizace **závisí** do značné míry na **vstupních datech**.

Na vstupu dostanete vektor složený z vektorů náhodně generovaných čísel.

Vaším úkolem je čísla v každém vektoru **sečíst** a tento součet vložit do vektoru s řešením na index odpovídající pořadí vektoru, který jste sčítali.

Paralelní suma vektoru

Doimplementujte metody v `SumsOfVectors.cpp` a zajistěte, že

1. Výpočet sum je paralelní a každá metoda vrací korektní výsledky
2. Metody využívají požadované způsoby paralelizace

Paralelní suma vektoru

Doimplementujte metody v `SumsOfVectors.cpp` a zajistěte, že

1. Výpočet sum je paralelní a každá metoda vrací korektní výsledky
2. Metody využívají požadované způsoby paralelizace

Za spravné výsledky na každé ze **čtyř** datových sad dostanete 2b.

Díky za pozornost!

Budeme rádi za Vaši zpětnou vazbu! →

https://forms.gle/yi7FWBEw3mxgnJ9P7